

# The Sx Library

---

for compiling and evaluating expressions

---

Copyright 1999-2006, United States Government as represented by the Administrator of the National Aeronautics and Space Administration. No copyright is claimed in the United States under Title 17, U.S. Code.

This software and documentation are controlled exports and may only be released to U.S. Citizens and appropriate Permanent Residents in the United States. If you have any questions with respect to this constraint contact the GSFC center export administrator, <Thomas.R.Weisz@nasa.gov>.

This product contains software from the Integrated Test and Operations System (ITOS), a satellite ground data system developed at the Goddard Space Flight Center in Greenbelt MD. See <<http://itos.gsfc.nasa.gov>> or e-mail <itos@itos.gsfc.nasa.gov> for additional information.

You may use this software for any purpose provided you agree to the following terms and conditions:

1. Redistributions of source code must retain the above copyright notice and this list of conditions.
2. Redistributions in binary form must reproduce the above copyright notice and this list of conditions in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement:

This product contains software from the Integrated Test and Operations System (ITOS), a satellite ground data system developed at the Goddard Space Flight Center in Greenbelt MD.

This software is provided "as is" without any warranty of any kind, either express, implied, or statutory, including, but not limited to, any warranty that the software will conform to specification, any implied warranties of merchantability, fitness for a particular purpose, and freedom from infringement and any warranty that the documentation will conform to their program or will be error free.

In no event shall NASA be liable for any damages, including, but not limited to, direct, indirect, special or consequential damages, arising out of, resulting from, or in any way connected with this software, whether or not based upon warranty, contract, tort, or otherwise, whether or not injury was sustained by persons or property or otherwise, and whether or not loss was sustained from or arose out of the results of, or use of, their software or services provided hereunder.

## 1 Introduction

The Sx library contains functions and macros for parsing and evaluating expressions, particularly in YACC grammars. The Sx library was originally developed for use in *stol*, a lex/yacc based interpreter for the *Spacecraft Test and Operations Language*.

All externally visible names (for variables, functions, structures, typedefs, and macros) begin with *Sx* or *SX* and are declared or defined in *Sx.h* (see see Appendix A [*Sx.h*], page 18). Programs that use the Sx library should `#include <Sx.h>` near the top of each appropriate source file.

The Sx library keeps track of the type of each operand. Operand types include date, double, int, string, symbol, time, and unsigned.

## 2 SX Library Functions

### 2.1 Initialization

```
void SxInit (warn, panic, symlookup, symfree, symname, Function
              symnamefree, syminfo, symgetv, symgetaltv, symsetv)
```

*SxInit* must be called before any of the other *Sx\** functions are used. The parameters are functions that glue the *Sx* library to the application:

```
int (*warn) (char *, ...) Function
int (*panic) (char *, ...) Function
```

*warn* and *panic* are the functions the *Sx* library will use to report errors. The first parameter to these functions is a *printf* style format string; this string specifies the number and type of the remaining arguments. “*printf*” may be used for these functions.

*warn* or *panic* may get called multiple times to generate a message; the ‘\n’ character marks the end of a message. One error may result in multiple messages.

*Stol* treats *warn* messages as operator errors. Examples of *warn* messages include

```
can't divide 47.2 by 0.0
ABS needs 1 argument; 3 were provided
```

and

```
don't know how to convert string "hi mom" to a date
```

*Stol* treats *panic* messages as severe errors. *panic* messages may be generated by any *Sx* function and indicate program bugs, malloc failure, or similarly severe conditions. Unlike *warn* messages, *panic* messages aren’t usually meaningful except to a programmer with access to the source code. Examples of *panic* messages include

```
SxNew: NEW(SxObjStruct) -> NULL
```

and

```
SxEval: obj->type = 255
```

Either (or both) of *warn* or *panic* may be *NULL*, in which case those messages simply aren’t generated.

```
void * (*symlookup) (char *name, void *container, int Function
                    *index, int *range)
```

```
void (*symfree) (void *sym) Function
```

*symlookup* is what the *Sx* library uses to determine whether or not *name* is in the symbol table. *symlookup* returns a handle to the symbol that can be used when getting or setting the symbol’s value.

The following shows how *container*, *index*, and *range* will be used:

```

symbol: name;                                /* $$ = symlookup($1,NULL,NULL,NULL) */
symbol: symbol '.' name                      /* $$ = symlookup($3,$1,NULL,NULL) */
symbol: symbol '[' expr ']';                 /* $$ = symlookup(NULL,$1,&$3,NULL) */
symbol: symbol '[' expr '..' expr ']';       /* $$ = symlookup(NULL,$1,&$3,&$5) */
symbol: symbol '[' '..' expr ']';            /* $$ = symlookup(NULL,$1,NULL,&$4) */
symbol: symbol '[' expr '..' ']';            /* $$ = symlookup(NULL,$1,&$3,NULL) */

```

*symlookup* generates an error message and returns NULL if the parameters don't identify a symbol.

*symfree* is either NULL or names the function that will free what *symlookup* returned. If *symfree* is not NULL, the Sx library will call *symfree* when it is finished with *sym*.

<code>char * (*symname) (void *sym)</code>	Function
<code>void (*symnamefree) (char *name)</code>	Function

*symnamefree* is either NULL or names the function that will free what *symname* returned. If *symnamefree* is not NULL, the Sx library will call *symnamefree* when it is finished with *name*.

<code>unsigned (*syminfo) (void *sym)</code>	Function
The value returned by <i>syminfo</i> is described in <i>Sx.h</i> .	

<code>SxObj (*symgetv) (SxObj obj, void *sym)</code>	Function
--	----------

<code>SxObj (*symgetaltv) (SxObj obj, void *sym)</code>	Function
---	----------

<code>int (*symsetv) (void *symbol, SxObj obj)</code>	Function
---	----------

*symgetv* is the function that retrieves *sym*'s value from the symbol table; *symgetaltv* is the function that retrieves *sym*'s converted ("P $\ell$ ") value from the symbol table; *symsetv* is the function that sets *sym*'s value.

*sym* is what *symlookup* returned.

If *symgetv* (or *symgetaltv*) is able to retrieve *sym*'s value, it sets *obj* to that value and returns *obj*. Otherwise *symgetv* must leave *obj* unchanged and return NULL.

If *symsetv* is able to set *sym*'s value it returns 0, otherwise it returns non-zero.

<code>SxErrhandType SxSetPanic (SxErrhandType newpanic)</code>	Function
--	----------

*SxSetPanic* changes the *panic* function to *newpanic* and returns the previous *panic* function.

<code>SxErrhandType SxSetWarn (SxErrhandType newwarn)</code>	Function
--	----------

*SxSetWarn* changes the *warn* function to *newwarn* and returns the previous *warn* function.

## 2.2 Parsing expressions

<code>SxObj SxParse (char *string)</code>	Function
---	----------

*SxParse* 'compiles' *string* and returns the *SxObj* that represents the compiled expression.

## 2.3 SX\_DATE objects and functions

*SX\_DATE* objects represent absolute moments in time, as opposed to *SX\_TIME* objects, which represent durations of time.

Internally, dates are stored in a *UNIX\_TIME* structure as the number of seconds since the UNIX epoch, the midnight that began Jan 1 1970.

In stol, dates have the form *yy-ddd-hh:mm:ss.uuuuuu* where *yy* is the two digit year, *ddd* the one to three digit julian day of year (Jan 1 is day 1), *hh* is the one or two digit hour (13 means 1 in the afternoon), *mm* is the one or two digit minute, *ss* is the one or two digit second, and *.uuuuuu* is the optional one to six digit fractional second. For example, *94-212-14:53:20.06* or *95-12-09:01:12*.

**SxObj SxNewDate (UNIX\_TIME \*date)** Function

*SxNewDate* creates a new *SX\_DATE* object with value *date*. Returns the new object, which will be *SX\_DATE* if everything worked or *SX\_NULL* otherwise.

**SxObj SxSetDate (SxObj obj, UNIX\_TIME \*date)** Function

*SxSetDate* assigns *date* to *obj*. Returns *obj*, which will be *SX\_DATE* if everything worked or *SX\_NULL* otherwise.

**int SxIsDate (SxObj obj)** Function

*SxIsDate* returns 1 if *obj* is *SX\_DATE*. Otherwise *SxIsDate* returns 0.

**UNIX\_TIME \* SxDates (SxObj obj)** Function

*SxDates* is a macro that references *obj*'s *UNIX\_TIME \* value*. For example, *SxDates(obj)->sec* references *obj*'s seconds field.

**SxObj SxConvertDate (SxObj obj)** Function

*SxConvertDate* converts primitive object (i.e., the result of *SxEval*) *obj* to *SX\_DATE*. If *obj* cannot be converted to *SX\_DATE*, *obj* is left unchanged and *NULL* is returned. Otherwise *obj* is returned.

When converting an integral or floating point value to a date, the value is treated as the number of seconds since the midnight that began Jan 1 1970. When converting a time value to a date, the time is treated as the number of seconds since the midnight that began Jan 1 1970.

## 2.4 SX\_DOUBLE functions

*SX\_DOUBLE* objects have *double* values. In stol, doubles look like 1.0, 2.3E4, .5 or 6E-7 (i.e., doubles have either a decimal point or an exponent).

**SxObj SxNewDouble (double d)** Function

*SxNewDouble* creates a new *SX\_DOUBLE* object.

**void SxSetDouble (SxObj *obj*, double *d*)** Function  
*SxSetDouble* assigns *d* to *obj*. Returns *obj*, which will be *SX\_DOUBLE* if everything worked and *SX\_NULL* otherwise.

**int SxIsDouble (SxObj *obj*)** Function  
*SxIsDouble* returns 1 if *obj* is *SX\_DOUBLE*. Otherwise *SxIsDouble* returns 0.

**double SxDouble (SxObj *varobj*)** Function  
*SxDouble* is a macro that references *obj*'s *double* value.

**SxObj SxConvertDouble (SxObj *obj*)** Function  
*SxConvertDouble* converts primative object (i.e., the result of *SxEval*) *obj* to *SX\_DOUBLE*. If *obj* cannot be converted to *SX\_DOUBLE*, *obj* is left unchanged and *NULL* is returned. Otherwise *obj* is returned.

## 2.5 SX\_EXPR objects and functions

**SxObj SxNewExpr (int *operator*, SxObj *operands*)** Function  
*SxNewExpr* creates a new *SX\_EXPR* object. *SxNewExpr* destroys *operands* (as if via *SxFree(operands)*). *operands* must be *SX\_LIST*.

**SxObj SxNewBinary (SxObj *arg1*, int *operator*, SxObj *arg2*)** Function  
*SxNewBinary* creates a new *SX\_EXPR* object and destroys *arg1* and *arg2*. The binary operators are '+', '-', '\*', '/', '^', *SX\_EQ*, *SX\_GE*, *SX\_GT*, *SX\_LT*, *SX\_NE*, and *SX\_XOR*. (*SX\_AND* and *SX\_OR* are used with *SxNewAndOr* – see below).

**SxObj SxNewUnary (int *operator*, SxObj *arg*)** Function  
*SxNewUnary* creates a new *SX\_EXPR* object and destroys *arg*. The unary operators are ' ', *SX\_NEGATE*, '^', *SX\_NOT*, and *SX\_P\_AT*.

**SxObj SxNewAndOr (SxObj *left*, int *andOr*, SxObj *right*)** Function  
*SxNewAndOr* creates a new *SX\_EXPR* object and destroys *left* and *right*. *andOr* is either *SX\_AND* or *SX\_OR*. *SX\_AND* and *SX\_OR* get their own function since, unlike the other binary operators, these are short-circuit operators (which means, for example, that the right side of an *SX\_AND* expression isn't evaluated unless the left side is true).

**SxObj SxNewFunc (char \**func*, SxObj *args*)** Function  
*SxNewFunc* creates a new *SX\_EXPR* object and destroys *args*. *args* must be *SX\_LIST*; *func* must be the name of a stol function. If *func* isn't the name of a stol function or if the number of arguments is wrong, *warn()* is called and an *SX\_NULL* object is returned.

**int SxIsExpr (SxObj *obj*)** Function  
*SxIsExpr* returns 1 if *obj* is *SX\_EXPR*. Otherwise *SxIsExpr* returns 0.

**SxExpr \* SxExpr (SxObj *obj*)** Function  
*SxExpr* is a macro that references *obj*'s *SxObjExpr* \* value.

## 2.6 SX\_EXPRNODE functions

*SX\_EXPRNODE* objects are used by the Sx library to describe subexpressions within an *SX\_EXPR* object.

**int SxIsExpn (SxObj *obj*)** Function  
*SxIsExpn* returns 1 if *obj* is *SX\_EXPRNODE*. Otherwise *SxIsExpn* returns 0.

**SxObjExpn \* SxExpn (SxObj *obj*)** Function  
*SxExpn* is a macro that references *obj*'s *SxObjExpn* \* value.

## 2.7 SX\_HEX functions

*SX\_HEX* objects have *SxObjHex* values. Stol uses *SX\_HEX* in the RAW and RAWTF directives.

**SxObj SxNewHex (unsigned byte)** Function  
*SxNewHex* creates a new *SX\_HEX* object with the one byte value *byte*.

**SxObj SxAppendHex (SxObj *obj*, unsigned byte)** Function  
*SxAppendHex* appends another byte to an *SX\_HEX* object. Returns *obj*.

**int SxIsHex (SxObj *obj*)** Function  
*SxIsHex* returns 1 if *obj* is *SX\_HEX*. Otherwise *SxIsHex* returns 0.

**SxObjHex \* SxHex (SxObj *varobj*)** Function  
*SxHex* is a macro that references *obj*'s *SxObjHex* value. For example, *SxHex(obj)->n* is the number of bytes in *SxHex(obj)->hex*.

## 2.8 SX\_INT functions

*SX\_INT* objects have *int* values. In stol, ints look like 123 (without a decimal point).

**SxObj SxNewInt (int *i*)** Function  
*SxNewInt* creates a new *SX\_INT* object with value *i*.

**void SxSetInt (SxObj *obj*, int *i*)** Function  
*SxSetInt* assigns *i* to *obj*. Returns *obj*, which will be *SX\_INT* if everything worked and *SX\_NULL* otherwise.

**int SxIsInt (SxObj *obj*)** Function  
*SxIsInt* returns 1 if *obj* is *SX\_INT*. Otherwise *SxIsInt* returns 0.

**int SxInt (SxObj *obj*)** Function  
*SxInt* is a macro that references *obj*'s *int* value.

**SxObj SxConvertInt (SxObj *obj*)** Function  
*SxConvertInt* converts primitive object (i.e., the result of *SxEval*) *obj* to *SX\_INT*. If *obj* cannot be converted to *SX\_INT*, *obj* is left unchanged and *NULL* is returned. Otherwise *obj* is returned.

## 2.9 SX\_LIST functions

**SxObj SxNewList (SxObj *item*)** Function  
*SxNewList* creates a new *SX\_LIST* object with one item *item* and destroys *item* (as if via *SxFree(item)*).

**SxObj SxAppendList (SxObj *list*, SxObj *item*)** Function  
*SxAppendList* appends *item* to *list* and destroys *item*. *SxAppendList* returns *list*.

**int SxCountList (SxObj *list*)** Function  
*SxCountList* counts the number of items in *list*. *list* must be *SX\_LIST*!

**int SxIsList (SxObj *obj*)** Function  
*SxIsList* returns 1 if *obj* is *SX\_LIST*. Otherwise, *SxIsList* returns 0.

**SxObjList \* SxList (SxObj *list*)** Function  
*SxList* is a macro that references *list*'s *SxObjList \** value.

Here's an example (in yacc) of how to create a list:

```
list: item      {$$ = SxRegister(SxNewList($1));};
list: list ',' item {$$ = SxAppendList($1,$3);};
```

And here's an example of how to traverse a list:

```
SxObjList *node;
...
node = SxList(list);
while (node) {
    do_something_with_obj(node->obj);
    node = node->next;
}
```

## 2.10 SX\_NULL objects and functions

*SX\_NULL* objects are objects without values.

**SxObj SxNew ()** Function  
*SxNew* creates a new *SX\_NULL* object.

**void SxMakeNull (SxObj obj)** Function  
*SxMakeNull* frees all memory utilized by *obj* and transforms *obj* to *SX\_NULL*.

**int SxIsNull (SxObj obj)** Function  
*SxIsNull* returns 1 if *obj* is *SX\_NULL*. Otherwise *SxIsNull* returns 0.

## 2.11 SX\_STRING functions

*SX\_STRING* objects have *char \** values.

**SxObj SxNewString (char \*s)** Function  
*SxNewString* creates a new *SX\_STRING* object with value *s*.

**int SxSetString (SxObj obj, char \*s)** Function  
*SxSetString* assigns *s* to *obj*. Returns *obj*, which will be *SX\_STRING* if everything worked and *SX\_NULL* otherwise.

**int SxAppendString (SxObj obj, char \*s)** Function  
*SxAppendString* concatenates *s* to the *obj*. *obj* must be *SX\_STRING*!

**int SxIsString (SxObj obj)** Function  
*SxIsString* returns 1 if *obj* is *SX\_STRING*. Otherwise *SxIsString* returns 0.

**char \* SxString (SxObj obj)** Function  
*SxString* references *obj*'s *char \** value.

**SxObj SxConvertString (SxObj obj)** Function  
*SxConvertString* converts primitive object (i.e., the result of *SxEval*) *obj* to *SX\_STRING*. If *obj* cannot be converted to *SX\_STRING*, *obj* is left unchanged and *NULL* is returned. Otherwise *obj* is returned.

## 2.12 SX\_SYMBOL functions

*SX\_SYMBOL* objects represent names in the symbol table and have *void \** values. See *SxInit* for how to glue the *Sx* library to the symbol table.

<code>SxObj SxNewSymbol (char *name)</code>	Function
<code>SxObj SxSymArray (SxObj sym, int index)</code>	Function
<code>SxObj SxSymSlice (SxObj sym, int *start, int *end)</code>	Function
<code>SxObj SxSymStruct (SxObj sym, char *member)</code>	Function

These functions create or alter *SX\_SYMBOL* objects. Here's how they're intended to be used:

```
sym: TokName
      {$$ = SxRegister(SxNewSymbol(SxString($1)));}
sym: sym '[' expr ']'
      {$$ = SxSymArray($1,SxEvalInt($3));}
sym: sym '[' expr TokDotDot expr ']'
      {int i = SxEvalInt($3), j=SxEvalInt)$5; $$ = SxSymSlice($1,&i,&j);}
sym: sym '.' TokName
      {$$ = SxSymStruct($1,SxString($3));}
```

<code>SxObj SxSymGetval ()</code>	Function
-----------------------------------	----------

<code>SxObj SxSymSetval()</code>	Function
----------------------------------	----------

<code>int SxIsSymbol (SxObj varobj)</code>	Function
<i>SxIsSymbol</i> returns 1 if <i>obj</i> is <i>SX_SYMBOL</i> . Otherwise, <i>SxIsSymbol</i> returns 0.	

<code>void * SxSymbol (SxObj obj)</code>	Function
<i>SxSymbol</i> references <i>obj</i> 's <i>void *</i> value.	

## 2.13 SX\_TIME functions

*SX\_TIME* objects have *UNIX\_TIME* values which represent relative (as opposed to absolute) times.

In stol, times have the form *hh:mm:ss.uuuuuu* where *hh*: is the optional one or more digit hour, *mm* is the one or two (more if *hh*: not specified) digit minute, *ss* is the one or two digit second, and *.uuuuuu* the optional one to six digit fractional second. For example, *1:00:00*, *1:30.0*, and *196:00:00*.

<code>SxObj SxNewTime (UNIX_TIME *time)</code>	Function
<i>SxNewTime</i> creates a new <i>SX_TIME</i> object with value <i>time</i> . <i>time</i> must not be NULL!	

<code>int SxSetTime (SxObj obj, UNIX_TIME *time)</code>	Function
<i>SxSetTime</i> assigns <i>time</i> to <i>obj</i> . Returns <i>obj</i> , which will be <i>SX_TIME</i> if everything worked and <i>SX_NULL</i> otherwise.	

**int SxIsTime (SxObj *obj*)** Function  
*SxIsTime* returns 1 if *obj* is *SX\_TIME*. Otherwise *SxIsTime* returns 0.

**UNIX\_TIME \* SxTime (SxObj *obj*)** Function  
*SxTime* is a macro that references *obj*'s *UNIX\_TIME* \* value.

**SxObj SxConvertTime (SxObj *obj*)** Function  
*SxConvertTime* converts primitive object (i.e., the result of *SxEval*) *obj* to *SX\_TIME*.  
If *obj* cannot be converted to *SX\_TIME*, *obj* is left unchanged and *NULL* is returned.  
Otherwise *obj* is returned.

## 2.14 SX\_UNKNOWN functions

*SX\_UNKNOWN* objects are intended to be used in YACC grammars that

```
#define YYSTYPE SxObj
```

*SX\_UNKNOWN* objects allow passing arbitrary structures via *YYSTYPE*.

Basically, an *SX\_UNKNOWN* object's value is a pointer to a *malloc*'d user defined object. In order for *SxFree* to work properly, the user defined object must be *malloc*'d and freeing the user defined object must free all space used by the object.

**SxObj SxNewUnknown (void \*ptr)** Function  
*SxNewUnknown* creates a new *SX\_UNKNOWN* object with value *ptr*, where *ptr* is either *NULL* or a value returned from *malloc()*.

**SxObj SxSetUnknown (SxObj *obj*, void \*ptr)** Function  
*SxSetUnknown* assigns *ptr* to *obj*, which will be *SX\_UNKNOWN* if everything worked and *SX\_NULL* otherwise.

**int SxIsUnknown (SxObj *obj*)** Function  
*SxIsUnknown* returns 1 if *obj* is *SxIsUnknown*. Otherwise *SxIsUnknown* returns 0.

**void \* SxUnknown (SxObj *obj*)** Function  
*SxUnknown* returns a pointer to an *SX\_UNKNOWN* object's user defined value.

## 2.15 SX\_UNSIGNED functions

*SX\_UNSIGNED* objects have *unsigned* values.

**SxObj SxNewUnsigned (unsigned *u*)** Function  
*SxNewUnsigned* creates a new *SX\_UNSIGNED* object with value *u*.

**SxObj SxSetUnsigned (SxObj *obj*, unsigned *u*)** Function  
*SxSetUnsigned* assigns *u* to *obj*. Returns *obj*, which will be *SX\_UNSIGNED* if everything worked and *SX\_NULL* otherwise.

**int SxIsUnsigned (SxObj obj)** Function  
*SxIsUnsigned* returns 1 if *obj* is *SX\_UNSIGNED*. Otherwise *ObjIsUnsigned* returns 0.

**int SxUnsigned (SxObj obj)** Function  
*SxUnsigned* is a macro that references *obj*'s *unsigned* value.

**SxObj SxConvertUnsigned (SxObj obj)** Function  
*SxConvertUnsigned* converts primative object (i.e., the result of *SxEval*) *obj* to *SX\_UNSIGNED*. If *obj* cannot be converted to *SX\_UNSIGNED*, *obj* is left unchanged and *NULL* is returned. Otherwise *obj* is returned.

## 2.16 Freeing objects

**int SxFree (SxObj obj)** Function  
*SxFree* frees all memory allocated to *obj* and destroys *obj*.

## 2.17 Registering objects

**SxObj SxRegister (SxObj obj)** Function  
*SxRegister* registers an object to be destroyed the next time *SxCleanup* is called. *obj* is the object to be registered. *SxRegister* returns *obj*.

**SxObj SxDupstroy (SxObj obj)** Function  
*SxDupstroy* returns an unregistered duplicate of *obj* and destroys *obj*. Hence the name, *duplicate and destroy*.

**int SxIsRegistered (SxObj obj)** Function  
*SxIsRegistered* returns 1 if *obj* is registered and returns 0 otherwise.

**void SxCleanup ()** Function  
*SxCleanup* *SxFree*'s all registered objects.

## Why register objects?

Consider the following yacc fragment:

```
pgm: expr '\n'      {printf("%d\n", SxEvalInt($1));};  

expr: int '+' int  {$$ = SxNewBinary($1,'+',$2);};  

int: NUMBER         {$$ = $1;};
```

Coded this way, it's full of memory leaks. In particular, there's no easy way to free objects created by incorrect input like *1+2+3+4#5* (when yacc hits the '#', it goes into error recovery and doesn't free the partial expression *1+2+3+4*).

The memory leak can be stopped by changing the above to:

```

pgm: {SxCleanup();} expr '\n' {printf("%d\n", SxEvalInt($1));}
expr: int '+' int   {$$ = SxRegister(SxNewBinary($1,'+',$2));}
int: NUMBER          {$$ = $1;};

```

It's important, when using *SxRegister*, to register objects as close as possible to their creation. Use

```

Pagename: expr {
    $$ = SxConvertString(SxEval($1,SxRegister(SxNew())));
}

```

instead of

```

Pagename: expr {
    $$ = SxRegister(SxConvertString(SxEval($1,SxNew())));
}

```

## 2.18 Evaluation to primitive object

**SxObj SxEval (SxObj *object*, SxObject *result*)** Function  
*SxEval* sets *result* a primitive object whose value is the result of the evaluation of *object*. The primitive objects are *SX\_DATE*, *SX\_DOUBLE*, *SX\_INT*, *SX\_NULL*, *SX\_STRING*, *SX\_TIME*, and *SX\_UNSIGNED*. If *object* is a primitive object, *result* gets set to a primitive object with the same type and value. If *object* is *SX\_SYMBOL* *result* gets set to a primitive object whose type and value depends on the value of symbol. If *object* is *SX\_EXPR* or *SX\_EXPRNODE* *result* gets set to a primitive object whose type and value depends on the value of the expression. *object* must not be *SX\_HEX*, *SX\_LIST*, or *SX\_UNKNOWN*.

If the evaluation was successful *result* is returned. Otherwise *result* gets set to *SX\_NULL* and *NULL* is returned. *SxEval* never changes *object*.

## 2.19 Evaluation to C type

**UNIX\_TIME \* SxEvalDate (SxObj *obj*)** Function  
*SxEvalDate* evaluates *obj*, converts the result to a date, and returns that date. Returns *NULL* if the evaluation or conversion failed. The returned date must be *free()*'d when no longer needed.

**double SxEvalDouble (SxObj *obj*)** Function  
*SxEvalDouble* evaluates *obj*, converts the result to a double, and returns that double. Returns 0.0 if the evaluation or conversion failed.

**int SxEvalInt (SxObj *obj*)** Function  
*SxEvalInt* evaluates *obj*, converts the result to an int, and returns that int. Returns 0 if the evaluation or conversion failed.

**char \* SxEvalString (SxObj obj)** Function

*SxEvalString* evaluates *obj*, converts the result to a string, and returns that string.

Returns NULL if the evaluation or conversion failed. The returned string must be *free()*'d when no longer needed.

**UNIX\_TIME \* SxEvalTime (SxObj obj)** Function

*SxEvalTime* evaluates *obj*, converts the result to a time, and returns that time. Returns NULL if the evaluation or conversion failed. The returned time must be *free()*'d when no longer needed.

**unsigned SxEvalUnsigned (SxObj obj)** Function

*SxEvalUnsigned* evaluates *obj*, converts the result to unsigned, and returns that unsigned. Returns 0 if the evaluation failed.

## 2.20 Miscellaneous functions

**char \* SxDescribe (SxObj obj)** Function

*SxDescribe* returns a static string that describes *obj*. For example, if *obj* is *SX-DATE*, *SxDescribe* might return the string "date 95-058-15:10:37.025700".

**int SxLooksLikeDate (char \*s)** Function

*SxLooksLikeDate* returns true (non-zero) if *s* looks like a date. For example, 94-327-16:36:19 looks like a date but 12:00:00 doesn't.

**int SxLooksLikeDouble (char \*s)** Function

*SxLooksLikeDouble* returns true (non-zero) if *s* looks like a double. For example, 1.0, .6, and 1e7 look like doubles but 17 doesn't.

**int SxLooksLikeTime (char \*s)** Function

*SxLooksLikeTime* returns true (non-zero) if *s* looks like a time. For example, 12:00:00 and 1:30.0 look like times but 30 and 95-12-08:00:00 don't.

## 3 Expressions

### 3.1 SX\_NOOP – Noop operator

*SX\_NOOP* is a unary operator that merely evaluates its argument. Stol uses it in *WAIT UNTIL* (*mnemonic*) to coerce *mnemonic* into *SX\_EXPR*.

### 3.2 SX\_NEGATE – Unary minus

*SX\_NEGATE* (- in stol) is a unary operator that computes the negative of its argument. If the argument evaluates to *SX\_TIME* or *SX\_DOUBLE*, or if the argument evaluates to an *SX\_STRING* that looks like a time or double value, the result will be *SX\_DOUBLE*. Otherwise, if the argument evaluates to anything that can be converted to *SX\_INT* the result will be *SX\_INT*. Otherwise, the negation fails.

### 3.3 ~ – Invert bits

The invert bits operator '~~'(~ in stol) is a unary operator that computes the bit inversion of its argument. If the argument evaluates to anything that can be converted to *SX\_UNSIGNED* the result will be *SX\_UNSIGNED*. Otherwise, the bitwise inversion fails.

### 3.4 + – Addition

The addition operator '+' (+ in stol) is a binary operator that computes the sum of its arguments. If either argument evaluates to *SX\_DATE* or an *SX\_STRING* that looks like a date, the other argument gets evaluated as *SX\_TIME* and the result of the addition is *SX\_DATE*. Otherwise, if either argument evaluates to *SX\_TIME* (or an *SX\_STRING* that looks like a time), the other argument gets evaluated as *SX\_TIME* and the result of the addition is *SX\_TIME*. Otherwise, if either argument evaluates to *SX\_DOUBLE* (or an *SX\_STRING* that looks like a double) the other argument gets evaluated as *SX\_DOUBLE* and the result of the addition is *SX\_DOUBLE*. Otherwise, if both arguments evaluate to *SX\_UNSIGNED* the result of the addition is *SX\_UNSIGNED*. Otherwise, both arguments get evaluated to *SX\_INT* and the result of the addition is *SX\_INT*.

### 3.5 - – Subtraction

The subtraction operator '-' (- in stol) is a binary operator that computes the difference of its arguments. If both arguments evaluate to *SX\_DATE* (or to *SX\_STRING*s that look like dates), the result of the subtraction is *SX\_TIME*. Otherwise, if either argument evaluates to *SX\_DATE* (or to an *SX\_STRING* that looks like a date), the other argument gets evaluated as *SX\_TIME* and the result of the subtraction is *SX\_DATE*. Otherwise, if either argument evaluates to *SX\_DOUBLE* (or to an *SX\_STRING* that looks like a double), the other argument gets evaluated as *SX\_DOUBLE* and the result of the subtraction is *SX\_DOUBLE*. Otherwise, if both arguments evaluate to *SX\_UNSIGNED* the result of the subtraction is *SX\_UNSIGNED* if the left argument is .GE. the right argument and

*SX\_INT* otherwise. Otherwise, both arguments get evaluated to *SX\_INT* and the result of the subtraction is *SX\_INT*.

### 3.6 \* and / – Multiplication and Division

The multiplication and division operators '\*' and '/' (\* and / in stol) are binary operators. For both operators: if either argument evaluates to *SX\_DOUBLE* or an *SX\_STRING* that looks like a double then both arguments get evaluated as *SX\_DOUBLE* and the result is *SX\_DOUBLE*. Otherwise both arguments get evaluated as *SX\_INT* and the result is *SX\_INT*.

Note that  $5/2$ , where 5 and 2 are both *SX\_INT*, evaluates to *SX\_INT* 2 rather than *SX\_DOUBLE* 2.5!

### 3.7 ^ – Exponentiation

The exponentiation operator '^' (^ in stol) is a binary operator that raises its left argument to the right argument power. Both arguments get evaluated as *SX\_DOUBLE* and the result of exponentiation is *SX\_DOUBLE*.

## 3.8 SX\_EQ, SX\_NE, SX\_GT, SX\_GE, SX\_LT, and SX\_LE

The relational operators *SX\_EQ*, *SX\_NE*, *SX\_GT*, *SX\_GE*, *SX\_LT*, and *SX\_LE* (.EQ. .NE.. .GT.. .GE.. .LT.. and .LE.. in stol) are binary operators that compare their arguments and evaluate to *SX\_INT* 1 if the relation is true and *SX\_INT* 0 if the relation is false.

If either argument evaluates to *SX\_DATE* and the other argument can be evaluated to either *SX\_DATE* or to an *SX\_STRING* that looks like a date, then both arguments get evaluated as *SX\_DATE* and a date comparison is performed.

Otherwise, if either argument evaluates to *SX\_TIME* and the other argument can be evaluated to either *SX\_TIME* or to an *SX\_STRING* that looks like a time, then both arguments get evaluated as *SX\_TIME* and a time comparison is performed.

Otherwise, if either argument evaluates to *SX\_DOUBLE* and the other argument can be evaluated to either *SX\_DOUBLE* or to an *SX\_STRING* that looks like a double, then both arguments get evaluated as *SX\_DOUBLE* and a floating point comparison is performed.

Otherwise, if both arguments evaluate to *SX\_UNSIGNED* then both arguments get evaluated as *SX\_UNSIGNED* and an unsigned comparison is performed.

Otherwise, if either argument evaluates to *SX\_INT* or *SX\_UNSIGNED* and the other argument can be evaluated as *SX\_INT*, then both arguments get evaluated as *SX\_INT* and an integer comparison is performed.

Otherwise, if both arguments can be evaluated as *SX\_STRING* then both arguments get evaluated as *SX\_STRING* and a string comparison is performed.

Otherwise, the relation is false.

(In particular, if either argument is *SX\_NULL* then the relation is false).

## 3.9 SX\_AND, SX\_OR, and SX\_XOR

The logical operators *SX\_AND*, *SX\_OR*, and *SX\_XOR* (.AND., .OR., and .XOR. in stol) are binary operators that evaluate to *SX\_INT* 0 (false) or 1 (true).

All three operators evaluate their arguments as *SX\_INT*. *SX\_AND* evaluates to 1 if both arguments are nonzero; *SX\_OR* evaluates to 1 if one or both arguments are nonzero; and *SX\_XOR* evaluates to 1 if either – but not both – argument is nonzero.

*SX\_AND* and *SX\_OR* are special in that these are short-circuit operators: unlike the other binary operators, *SX\_AND* and *SX\_OR* evaluate and examine the left argument's value before evaluating the right argument. For *SX\_AND*, if the left argument is 0, the right argument doesn't need to be (and isn't) evaluated. Similarly, for *SX\_OR*, if the left argument is 1, the right argument doesn't need to be (and isn't) evaluated.

## 3.10 SX\_NOT – Logical negation

*SX\_NOT* (.NOT. in stol) is a unary operator that computes the logical negation of its argument. If the argument evaluates to anything that can be converted to *SX\_INT* the result will be *SX\_INT* (0 if the argument evaluates to a non-zero value and 1 if the argument evaluates to 0). Otherwise, the logical negation fails.

## 3.11 P@ operator

The *P@* operator is a unary operator that returns the converted value of its *SX\_SYMBOL* argument.

## 3.12 STOL functions

See section “STOL Functions” in *STOL Functions* for the list of functions that the *Sx* library evaluates.

## 4 Bugs

- The Sx library uses `pow()` (see *man 3m pow*) and defines `matherr()`. This may conflict with other definitions of `matherr()`. Sorry.

## Appendix A Sx.h

```

/* $Id: Sx.h,v 3.9 2006/07/24 21:17:20 bgoldman Exp $
** Author: Tim Singletary
*/
#ifndef SX_H
#define SX_H

#include "AVL.h"
#include "unix_time.h"
#include "dbif.h"

/*****************/
/* Data structures */
/*****************/

typedef int (*SxErrhandType)(char *, ...);

typedef struct SxObj* SxObj;
typedef struct SxObjExpn SxObjExpn;
typedef struct SxObjExpr SxObjExpr;
typedef struct SxObjHex SxObjHex;
typedef struct SxObjList SxObjList;

typedef enum
    SX_DATE, SX_DOUBLE, SX_EXPR, SX_EXPRNODE, SX_HEX, SX_INT, SX_LIST,
    SX_NULL, SX_STRING, SX_SYMBOL, SX_TIME, SX_UNKNOWN, SX_UNSIGNED,
    SxObjType;
    ;

struct SxObjList
    SxObj obj;
    SxObjList *next;
;

struct SxObjExpn
    int operator;
    SxObjList *operands;
;

struct SxObjExpr
    SxObjExpn *root;
    int n_symbols;
# define SX_MAX_EXPR_SYMBOLS 100
    void *list[SX_MAX_EXPR_SYMBOLS+1];
;

struct SxObjHex
;
```

```

        int n;
#define SX_MAX_HEX 1200
        unsigned char hex[SX_MAX_HEX];
;

struct SxObj
{
    SxObjType type;
    union
    {
        void* ptr;
        UNIX_TIME* sx_date;
        double* sx_double;
        SxObjExpr* sx_expr;
        SxObjExpn* sx_exprnode;
        SxObjHex* sx_hex;
        int sx_int;
        SxObjList* sx_list;
        char* sx_string;
        void* sx_symbol;
        UNIX_TIME* sx_time;
        void* sx_unknown;
        unsigned sx_unsigned;
        value;
    };
};

struct SxFns
{
    SxErrhandType panic;
    SxErrhandType warn;
    void* (*symlookup)(char *name, void *container, int *index, int *range);
    void (*symfree)(void *sym);
    char* (*symname)(void *sym);
    void (*symnamefree)(char *name);
    SxObj (*symgetv)(SxObj obj, void *sym);
    SxObj (*symgetaltv)(SxObj obj, void *sym);
    int (*symsetv)(void *sym, SxObj obj);
    unsigned (*syminfo)(void *sym);
};

extern struct SxFns Sx;

/* Sx.syminfo() returns an 'unsigned' where each bit is a flag.
   The flags are: */

/*      One (and only one) of these must be set: */
#define SX_SYMINFO_ISGLOBAL (1 << 0)
#define SX_SYMINFO_ISLOCAL (1 << 1)
#define SX_SYMINFO_ISCMD (1 << 2)
#define SX_SYMINFO_ISMNEM (1 << 3)

/*      This bit is set only if the symbol is a slice or array with

```

```

        more than one element. */
#define SX_SYMINFO_ISARRAY      (1 << 4)
/* And this bit is set only if the symbol is a structure. */
#define SX_SYMINFO_ISSTRUCT     (1 << 5)
/* Note that a symbol can be both array and structure! */

/* These bits are never set unless the symber is a mnemonic. */
#define SX_SYMINFO_EXISTS       (1 << 6)
#define SX_SYMINFO_ISSTATIC      (1 << 7)
#define SX_SYMINFO_REDHI         (1 << 8)
#define SX_SYMINFO_REDLO         (1 << 9)
#define SX_SYMINFO_INLIMITS      (1 << 10)
#define SX_SYMINFO_YELLOWLO      (1 << 11)
#define SX_SYMINFO_YELLOWHI      (1 << 12)

#define SX_AND      10
#define SX_EQ       20
#define SX_GE       30
#define SX_GT       40
#define SX_LE       50
#define SX_LT       60
#define SX_NE       70
#define SX_NEGATE   80
#define SX_NOOP     90
#define SX_NOT      100
#define SX_OR       110
#define SX_P_AT     120
#define SX_XOR      130

extern int sx_matherr_flag;

/*****************/
/* Functions and macros */
/*****************/

SxObj      SxAppendHex(SxObj, unsigned);
SxObj      SxAppendList(SxObj, SxObj);
SxObj      SxAppendString(SxObj, char *);
void       SxCleanup(void);
SxObj      SxConvertDate(SxObj);
SxObj      SxConvertDouble(SxObj);
SxObj      SxConvertInt(SxObj);
SxObj      SxConvertString(SxObj);
SxObj      SxConvertTime(SxObj);
SxObj      SxConvertUnsigned(SxObj);
int        SxCountList(SxObj);

```

```

#define      SxDates(obj)        /* UNIX_TIME */ ((obj)->value.sx_date)
char *
#define      SxDescrbe(SxObj);   /* double */ (*((obj)->value.sx_double))
SxObj
#define      SxDupstroy(SxObj);
SxObj
#define      SxEval(SxObj, SxObj);
UNIX_TIME *
double      SxEvalDate(SxObj);
void       SxEvalDouble(SxObj);
void       SxEvalInit();
int        SxEvalInt(SxObj);
char *
UNIX_TIME * SxEvalString(SxObj);
unsigned    SxEvalTime(SxObj);
#define      SxEvalUnsigned(SxObj);
#define      SxExpr(obj)         /* SxObjExpr */ ((obj)->value.sx_expr)
#define      SxExpn(obj)         /* SxObjExpn */ ((obj)->value.sx_exprnode)
void       SxFree(SxObj);
void SxFuncEval(int, SxObjList *, SxObj);
#define      SxHex(obj)          /* SxObjHex */ ((obj)->value.sx_hex)
#define      SxInt(obj)          /* int */ ((obj)->value.sx_int)
void       SxInit(SxErrhandType, SxErrhandType, /* warn, panic */
                  void * (*)(char *, void *, int *, int *), /* symlookup */
                  void * (*)(void *), /* symfree */
                  char * (*)(void *), /* symname */
                  void * (*)(char *), /* symnamefree */
                  unsigned * (*)(void *), /* syminfo */
                  SxObj * (*)(SxObj, void *), /* symgetv */
                  SxObj * (*)(SxObj, void *), /* symgetaltv */
                  int * (*)(void *, SxObj)); /* symsetv */
#define      SxIsDate(obj)        ((obj) && (obj)->type == SX_DATE)
#define      SxIsDouble(obj)      ((obj) && (obj)->type == SX_DOUBLE)
#define      SxIsExpr(obj)        ((obj) && (obj)->type == SX_EXPR)
#define      SxIsExprnode(obj)    ((obj) && (obj)->type == SX_EXPRNODE)
#define      SxIsHex(obj)         ((obj) && (obj)->type == SX_HEX)
#define      SxIsInt(obj)         ((obj) && (obj)->type == SX_INT)
#define      SxIsList(obj)        ((obj) && (obj)->type == SX_LIST)
#define      SxIsNull(obj)        ((obj) && (obj)->type == SX_NULL)
#define      SxIsNumber(obj)      ((obj) && (((obj)->type == SX_INT) \
|| ((obj)->type == SX_UNSIGNED) \
|| ((obj)->type == SX_DOUBLE)))
#define      SxIsString(obj)      ((obj) && (obj)->type == SX_STRING)
#define      SxIsSymbol(obj)      ((obj) && (obj)->type == SX_SYMBOL)
#define      SxIsTime(obj)        ((obj) && (obj)->type == SX_TIME)
#define      SxIsUnknown(obj)     ((obj) && (obj)->type == SX_UNKNOWN)
#define      SxIsUnsigned(obj)    ((obj) && (obj)->type == SX_UNSIGNED)
#define      SxList(obj)          /* SxObjList */ ((obj)->value.sx_list)
int        SxLooksLikeDate(char *);
int        SxLooksLikeDouble(char *);
int        SxLooksLikeTime(char *);
void SxMakeNull(SxObj);

```

```

SxObj      SxNew();
SxObj      SxNewAndOr(SxObj, int, SxObj);
SxObj      SxNewBinary(SxObj, int, SxObj);
SxObj      SxNewDate(UNIX_TIME *);
SxObj      SxNewDouble(double);
SxObj      SxNewExpr(int, SxObj);
SxObj  SxNewFunc(char *, SxObj);
SxObj      SxNewHex(unsigned);
SxObj      SxNewInt(int);
SxObj      SxNewList(SxObj);
SxObj      SxNewString(char *);
SxObj      SxNewSymbol(char *);
SxObj      SxNewTime(UNIX_TIME *);
SxObj      SxNewUnary(int, SxObj);
SxObj  SxNewUnknown(void *);
SxObj      SxNewUnsigned(unsigned);
void       SxObjInit(
            void * (*)(char *, void *, int *, int *), /* symlookup */
            void * (*)(void *), /* symfree */
            char * (*)(void *), /* symname */
            void * (*)(char *), /* symnamefree */
            unsigned * (*)(void *), /* syminfo */
            SxObj * (SxObj, void *), /* symgetv */
            SxObj * (SxObj, void *), /* symgetaltv */
            int * (void *, SxObj)); /* symsetv */
SxObj  SxParse(char *);
SxObj      SxRegister(SxObj);
SxObj      SxSetDate(SxObj, UNIX_TIME *);
SxObj      SxSetDouble(SxObj, double);
SxObj      SxSetInt(SxObj, int);
SxErrhandType  SxSetPanic(SxErrhandType);
SxObj  SxSetString(SxObj, char *);
SxObj      SxSetTime(SxObj, UNIX_TIME *);
SxObj  SxSetUnknown(SxObj, void *);
SxObj      SxSetUnsigned(SxObj, unsigned);
SxErrhandType  SxSetWarn(SxErrhandType);
#define      SxString(obj)      /* char * */ ((obj)->value.sx_string)
SxObj      SxSymArray(char *, int);
#define      SxSymbol(obj)      /* void * */ ((obj)->value.sx_symbol)
SxObj      SxSymSlice(SxObj, int, int );
SxObj      SxSymStruct(SxObj, char *);
#define      SxTime(obj)        /* UNIX_TIME * */ ((obj)->value.sx_time)
#define      SxUnknown(obj)     /* void * */ ((obj)->value.sx_unknown)
#define      SxUnsigned(obj)    /* unsigned */ ((obj)->value.sx_unsigned)

#endif

```

# Index to functions

(	SxIsDouble .....	5
(*panic) .....	SxIsExpn .....	6
(*symfree) .....	SxIsExpr .....	6
(*symgetaltv) .....	SxIsHex .....	6
(*symgetv) .....	SxIsInt .....	7
(*syminfo) .....	SxIsList .....	7
(*symlookup) .....	SxIsNull .....	8
(*symname) .....	SxIsRegistered .....	11
(*symnamefree) .....	SxIsString .....	8
(*symsetv) .....	SxIsSymbol .....	9
(*warn) .....	SxIsTime .....	10
	SxIsUnknown .....	10
	SxIsUnsigned .....	11
<b>M</b>	SxList .....	7
matherr .....	SxLooksLikeDate .....	13
	SxLooksLikeDouble .....	13
	SxLooksLikeTime .....	13
<b>P</b>	SxMakeNull .....	8
panic .....	SxNew .....	8
	SxNewAndOr .....	5
	SxNewBinary .....	5
	SxNewDate .....	4
<b>S</b>	SxNewDouble .....	4
SxAppendHex .....	SxNewExpr .....	5
SxAppendList .....	SxNewFunc .....	5
SxAppendString .....	SxNewHex .....	6
SxCleanup .....	SxNewInt .....	6
SxConvertDate .....	SxNewList .....	7
SxConvertDouble .....	SxNewString .....	8
SxConvertInt .....	SxNewSymbol .....	9
SxConvertString .....	SxNewTime .....	9
SxConvertTime .....	SxNewUnary .....	5
SxConvertUnsigned .....	SxNewUnknown .....	10
SxCountList .....	SxNewUnsigned .....	10
SxDate .....	SxParse .....	3
SxDescribe .....	SxRegister .....	11
SxDouble .....	SxSetDate .....	4
SxDupstroy .....	SxSetDouble .....	5
SxEval .....	SxSetInt .....	6
SxEvalDate .....	SxSetPanic .....	3
SxEvalDouble .....	SxSetString .....	8
SxEvalInt .....	SxSetTime .....	9
SxEvalString .....	SxSetUnknown .....	10
SxEvalTime .....	SxSetUnsigned .....	10
SxEvalUnsigned .....	SxSetWarn .....	3
SxExpn .....	SxString .....	8
SxExpr .....	SxSymArray .....	9
SxFree .....	SxSymbol .....	9
SxHex .....	SxSymGetval .....	9
SxInit .....	SxSymSetval() .....	9
SxInt .....	SxSymSlice .....	9
SxIsDate .....		

SxSymStruct .....	9	symlookup.....	2
SxTime.....	10	symname.....	2
SxUnknown .....	10	symnamefree .....	2
SxUnsigned .....	11	symsetv.....	2
symaltgetv .....	2		
symfree.....	2		
symgetv.....	2		
syminfo.....	2	W	
		warn.....	2

# Table of Contents

<b>1</b>	<b>Introduction . . . . .</b>	<b>1</b>
<b>2</b>	<b>SX Library Functions . . . . .</b>	<b>2</b>
2.1	Initialization . . . . .	2
2.2	Parsing expressions . . . . .	3
2.3	SX_DATE objects and functions . . . . .	4
2.4	SX_DOUBLE functions . . . . .	4
2.5	SX_EXPR objects and functions . . . . .	5
2.6	SX_EXPRNODE functions . . . . .	6
2.7	SX_HEX functions . . . . .	6
2.8	SX_INT functions . . . . .	6
2.9	SX_LIST functions . . . . .	7
2.10	SX_NULL objects and functions . . . . .	8
2.11	SX_STRING functions . . . . .	8
2.12	SX_SYMBOL functions . . . . .	9
2.13	SX_TIME functions . . . . .	9
2.14	SX_UNKNOWN functions . . . . .	10
2.15	SX_UNSIGNED functions . . . . .	10
2.16	Freeing objects . . . . .	11
2.17	Registering objects . . . . .	11
Why register objects? . . . . .	11	
2.18	Evaluation to primitive object . . . . .	12
2.19	Evaluation to C type . . . . .	12
2.20	Miscellaneous functions . . . . .	13
<b>3</b>	<b>Expressions . . . . .</b>	<b>14</b>
3.1	SX_NOOP – Noop operator . . . . .	14
3.2	SX_NEGATE – Unary minus . . . . .	14
3.3	~ – Invert bits . . . . .	14
3.4	+ – Addition . . . . .	14
3.5	- – Subtraction . . . . .	14
3.6	* and / – Multiplication and Division . . . . .	15
3.7	^ – Exponentiation . . . . .	15
3.8	SX_EQ, SX_NE, SX_GT, SX_GE, SX_LT, and SX_LE . . . . .	15
3.9	SX_AND, SX_OR, and SX_XOR . . . . .	16
3.10	SX_NOT – Logical negation . . . . .	16
3.11	P@ operator . . . . .	16
3.12	STOL functions . . . . .	16
<b>4</b>	<b>Bugs . . . . .</b>	<b>17</b>

**Appendix A Sx.h ..... 18**

**Index to functions ..... 23**